ReqD Notes

Software travelogue

Oct 2, 1990

In order to present a compatible network interface to the accelerator Vax computers, extensive changes are being made in the Local Station software. The Vax assumes the use of a word-size node number, whereas the local station software has always used a byte-size node number. This assumption shows up in numerous parts of the software, and it even affects the page applications as well, as many of them allow input of a node number as a two hex digit field. Support for short idents must be retired, since there is no room for a word size node number in a short ident.

A key module in the system that is severely affected by this change is `ReqD`. This is the routine that a user program calls to make a data request and also the one which supports the data server request, all using the Classic protocol that was designed in 1980 and has evolved somewhat in recent years. This module was written in a style that has since been abandoned, and revisiting the code requires a rewrite in conform to more recently established practices. The previous style involved extensive use of registers with long-term significance, thus avoiding the creation of a local stack frame to contain local variables. It made the program logic extremely difficult to read. Its main virtue has been that it worked. In rewriting this code, a stack frame will be used and the practice of using registers whose significance lasts over pages of code will be resisted. Furthermore, the registers (other than `D0-D1/A0-A1`) will be preserved, thus making it compatible with most hi level language compiler register usage conventions.

The main purpose of these notes is to describe in some detail what the code does for internal documentation purposes. Its use is expected to be limited, however, as there is long term interest in replacing the Classic protocol support with the support for the D0 protocol, as the latter protocol is a logical extension of the Classic protocol and removes many of the limitations of that original data request protocol. The actual retiring of the Classic protocol is not expected for some time, however, as it will affect a number of other users of the local stations from several other platforms. Those users will need to adapt to use of the D0 protocol first. Also, the data server support by the local station will have to be given the D0 protocol before it can supplant the Classic protocol.

Overview

The routine `ReqData` is called by a user program to initiate a data request. The calling sequence is as follows:

```
Procedure ReqData(list: Byte; freq: Byte;
      nLtypes: Byte; VAR listypes: Integer;
      nIdents: Integer; VAR idents: Integer);
```

The list is a request-id chosen by the caller that is used to identify the request in the subsequent call to retrieve the data and to cancel the request. Values in the range 0–13 are allowed, depending upon the structure of the `LISTP` system table; it may be modified to extend the range somewhat. See more on this later.

The `freq` byte is actually a period count of 15 Hz cycles to specify a repetitive request. Zero means a one-shot request. The maximum value of 255 is therefore about 17 seconds.

The number of listypes, `nLtypes`, in the range 1–15, specifies the length of the listype array, specified by `listypes`. For each listype and associated #bytes value, the array of idents is processed to produce the resultant answer data. This means that the listypes must be ident-compatible; *i.e.*, they all must use the same ident type. Thus, it is not possible to combine a request for analog channel reading data with memory data, for example, in a single data request. (This is one of the limitations that the D0 protocol removes.) On the other hand, one may ask for readings, settings, nominals and tolerances for the same set of analog channels quite efficiently.

The number of idents in the array idents is given by `nIdents`. The hi nibble is used to hold the length of each ident. For example, an analog channel ident is composed of two words, the node# and the channel index, so the ident length in that case is 4 bytes. If the ident length is given as zero, the system assumes a default value. (The current default value is that for a short ident, but that will be changed soon to the value for a long ident.) In the future, the ident length may be required to be specified to open the door to alternate forms of idents for a given listype, such as using a name, for example. Of course, the lengths supported would have to differ to be distinguishable.

Since each ident carries within it a node#, one may make a request for data that refers to a number of different nodes, including the local node. The support software separates out the local idents from the external ones and issues a network request for the external ones. All this use of the network is made transparent to the user. S/he only has to issue the following call to collect the answers:

```
Procedure Collect(list: Byte; VAR status: Integer;
                  VAR answers: Integer);
```

The list number is the same small value used in the call to `ReqData`. An error status word is returned, where zero means no errors. The answer data is collected from the local node and/or external node answer buffers that were allocated when the request was initialized. The order and layout of the answer data is specified by the order of the request. The data for all the idents of each listype is separately padded to an even #bytes, in case both the #idents and the #bytes requested/ident were odd.

The system monitors the arrival of external answer fragments that are received from each external node participating in the request. If a node is tardy in returning the answer data, the `Collect` routine attempts to await the return of that node's data using a time-out of about 50 msec past the start of the cycle, after which it returns an error status of either 7 or 8. The value of 8 is used if no answer fragment has been received from that node since the request was initialized and 7 otherwise. If repetitive calls to `Collect` are made, and the tardiness is persistent over 2 seconds, the system reissues the data request to that node, hopeful that the node will revive and begin participation in the request.

To cancel a data request, use the following call:

```
Procedure Delist(list: Byte);
```

Again, the list argument is the same small value used in the `ReqData` call. If the request included external nodes, then a cancel message is sent to those nodes to cause them to cease delivery of answer fragments.

Internal list# logic

The `LISTP` table maintains a record of list#s in use, and it provides a means of avoiding reuse of the same list# in a data request for a period of time after a request has been cancelled. This is to prevent misinterpretation of answers to a new request issued immediately following a cancel of a previous request.

The `LISTP` table is divided into a "short set" and a "main set" of entries. The short set is indexed by small values (currently in the range 0–13 as noted above). A short set entry contains a "full list#" that is allocated from the main set. The main set is indexed by a full list#. Its entry contains a pointer to the request memory block (allocated from dynamic memory) that supports the data request while it is active. By maintaining a record of the last-used main set entry and a usage count for each main set entry, and by including some bits of the usage counter in the allocated list#, a newly-freed list# will not be reused for a long time.

It is important that a data request does not use the same list# as one which is already in use. The above `LISTP` logic can provide such service, but the user interface calls do not allow it. Fortunately, the only data requesting programs have heretofore been page applications, and only one of them can be active at one time. If more tasks need to make data requests, it might be necessary to provide user interface calls which *return* the full list# when the request is initialized and which *accept* that full list# in the `Collect` and `Delist` calls. The short set was designed to retain use of the present interface routines but still prevent the above misinterpretation of answers to new requests.

Delist
Call `GetListN` to get the full list# associated with the small list# argument. If it is valid, cancel the request by calling `Delist1`, and clear the short set entry by calling `SetListN`; otherwise, simply return.

Delist1
Call `GetListP` to get a pointer to the request memory block. Clear the main set entry by calling `SetListP`. Delete the request from the chain of active data requests. Capture the pointers to any external request block and/or a total answers block. Release the allocated request block memory. If there was an external request block, queue a cancel message to the network using the same destination node (which could have been a multicast address) that was used in making the external data request originally, and release the external request block. If there was a total answers block, release it also. `Delist1` preserves all registers. Its single argument is the full list# in `D0`.

Data Server requests
A data server request is one which originates from another node on the network. When a data request message is received that specifies the use of the data server, it is supported by the system on behalf of the network requester. The `Server` Task, which runs every cycle at about 40 msec into the cycle, scans the chain of active data requests for data server requests, calls `Collect` to retrieve the data (without waiting), and it queues the resulting "total answers" to the network requesting node. A total answers memory block, referenced by a field in the request block, carries the answer response to the network.

ReqDataS
This entry point is used by the `Network` Task when it receives a data server request. If the full list# (specified in the network request by the requesting node) is the same as one found in the total answers block of a currently-active data server request, then that request is cancelled. A new list# is obtained via `NewListN`. Note that the requesting node's list# is not used, since we cannot guarantee that it would not conflict with a currently-active `LISTP` entry. However, the original list# is retained for inclusion in the total answers response to the requesting node.

ReqData

The call is converted into a `ReqDataS`-compatible call by appending an additional zero argument to stand for the requesting node# argument that is the last argument placed on the stack for `ReqDataS`. The small list# argument is converted into a full list# via `GetListN`. If it is active, then the short set entry is cleared via `SetListN`. A new list# is obtained via `NewListN`, and is recorded in the short set entry via `SetListN`.

ReqdCom

This code is common to both `ReqData` and `ReqDataS`. Arguments `nLtypes` and `nIdents` are checked against reasonable ranges. The listypes are checked for being ident-compatible.

The number of bytes needed for an external request block and for the main request block is evaluated by scanning the idents for the number of idents from each external node represented in the request. Memory is allocated both for the request block and for an external request block, if needed, and the blocks initialized.

The pointer-type routines are called for each listype to translate each local ident into an internal pointer and each external ident into a reference to the predictable part of the external node's external answer buffer where the answers will be placed when the answer fragment message is received from that external node. The result of this "compilation" is an array of "internal pointers" that are interpreted at data request fulfillment time (via `Collect`) by read-type routines. This interpretation loop is optimized for speed, as data request fulfillment may be done for a number of active requests at 15 Hz.

The "`Age`" and "`Cntr`" fields in the external answer pointers are initialized for detection of tardy external nodes. The main set list# is established via `SetListP`.

If the request was a data server request, a total answers block is allocated and initialized for later queuing of the total answers to the original requester.

The new request is connected into the chain of active data requests via `InsChain`, which places it adjacent to another active data request from the same node, if there is any.

If there is an external request block, it is queued to the network via `OUTPQX`.

Note that ownership of each of the three memory blocks is assigned to `QMonitor` Task via `Assign`, as `QMonitor` may likely be the one which may have to release the memory when the request is cancelled. If the user calls `Delist` to cancel a request, a check is made to see whether the external memory block or the total answers memory block has been queued to the network but not actually yet transmitted. In that case, a flag is set for `QMonitor` to free the memory when the message has been transmitted. Otherwise, the ownership of those blocks is reacquired via `Grab` so that the blocks can be freed immediately.